Decorators

A Powerful Weapon in your Python Arsenal

Colton Myers

@basepi

http://bit.ly/dec-pycon-2014

SALTSTACK

Come visit our booth!

Open space tonight

Room 523A

17:00-19:00

Salt Sprint on Monday

The Plan

- What is a decorator?
- How are decorators constructed?
- How do they work?
- The “right” way to make decorators
- Examples

What is a decorator?

```
@my_decorator
def my_awesome_function():
    pass
```

Decorators wrap functions

Decorators wrap functions

- Add functionality
- Modify behavior
- Perform setup/teardown
- Diagnostics (timing, etc)

But first…

What is a function?

Everything in Python is an object.

Functions are objects

```
def myfunc():
    print(‘hello!’)
```

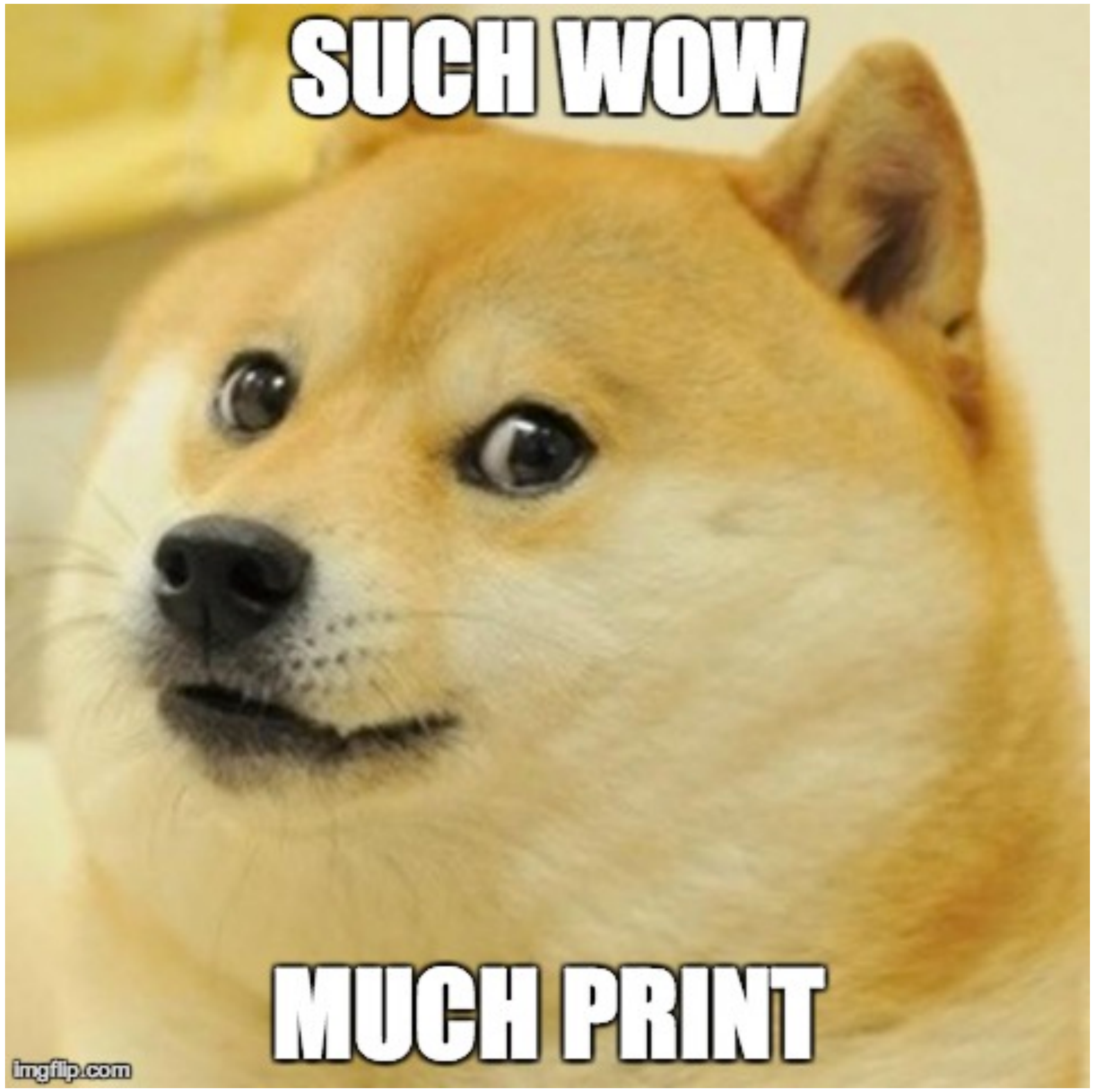
```
def myfunc():
    print(‘hello!’)

myfunc() #prints ‘‘hello!’’
f = myfunc
f()        #prints ‘‘hello!’’
```

Functions can create other functions

```
def make_printer(word):
    def inner():
        print(word)
    return inner

p = make_printer(‘such wow!’)
p() #prints ‘‘such wow!’’
```


SUCH WOW
MUCH PRINT
imgflip.com

This is called a closure

```
def make_printer(word):
    def inner():
        print(word)
    return inner
```

```
def make_printer(word):
    def inner():
        print(word)
    return inner
```

Decorators are closures

Decorators are closures

…usually.

(Decorators can also be created with classes)

```
def make_printer(word):
    def inner():
        print(word)
    return inner
```

```
def my_decorator(wrapped):
    def inner(*args, **kwargs):
        return wrapped(*args, **kwargs)
    return inner
```

```
def my_decorator(wrapped):
    def inner(*args, **kwargs):
        return wrapped(*args, **kwargs)
    return inner


@my_decorator
def myfunc():
    pass
```

```
def my_decorator(wrapped):
    def inner(*args, **kwargs):
        return wrapped(*args, **kwargs)
    return inner


def myfunc():
    pass
myfunc = my_decorator(myfunc)
```

```
def my_decorator(wrapped):
    def inner(*args, **kwargs):
        return wrapped(*args, **kwargs)
    return inner


@my_decorator
def myfunc():
    pass
```

```
def shout(wrapped):
    def inner(*args, **kwargs):
        print(‘BEFORE!’)
        ret = wrapped(*args, **kwargs)
        print(‘AFTER!’)
        return ret
    return inner

@shout
def myfunc():
    print(‘such wow!’)
```

```
def shout(wrapped):
    def inner(*args, **kwargs):
        print('BEFORE!')
        ret = wrapped(*args, **kwargs)
        print('AFTER!')
        return ret
    return inner

@shout
def myfunc():
    print('such wow!')
```

```
def shout(wrapped):
    def inner(*args, **kwargs):
        print(‘BEFORE!’)
        ret = wrapped(*args, **kwargs)
        print(‘AFTER!’)
        return ret
    return inner

@shout
def myfunc():
    print(‘such wow!’)
```

```
>>> myfunc()
BEFORE!
such wow!
AFTER!
```

How does this work?

```
def shout(wrapped):
    def inner(*args, **kwargs):
        print(‘BEFORE!’)
        ret = wrapped(*args, **kwargs)
        print(‘AFTER!’)
        return ret
    return inner

@shout
def myfunc():
    print(‘such wow!’)
```

```
def shout(wrapped):
    def inner(*args, **kwargs):
        print(‘BEFORE!’)
        ret = wrapped(*args, **kwargs)
        print(‘AFTER!’)
        return ret
    return inner

def myfunc():
    print(‘such wow!’)
myfunc = shout(myfunc)
```

```
def shout(wrapped):
    def inner(*args, **kwargs):
        print(‘BEFORE!’)
        ret = wrapped(*args, **kwargs)
        print(‘AFTER!’)
        return ret
    return inner

def myfunc():
    print(‘such wow!’)
myfunc = shout(myfunc)
```

```
def shout(wrapped):
    def inner(*args, **kwargs):
        print('BEFORE!')
        ret = wrapped(*args, **kwargs)
        print('AFTER!')
        return ret
    return inner

def myfunc():
    print('such wow!')
myfunc = shout(myfunc)
```

```
def shout(wrapped):
    def inner(*args, **kwargs):
        print('BEFORE!')
        ret = wrapped(*args, **kwargs)
        print('AFTER!')
        return ret
    return inner

def myfunc():
    print('such wow!')
myfunc = shout(myfunc)
```

```
def shout(wrapped):
    def inner(*args, **kwargs):
        print(‘BEFORE!’)
        ret = wrapped(*args, **kwargs)
        print(‘AFTER!’)
        return ret
    return inner

def myfunc():
    print(‘such wow!’)
myfunc = shout(myfunc)
```

```
def shout(wrapped):
    def inner(*args, **kwargs):
        print('BEFORE!')
        ret = wrapped(*args, **kwargs)
        print('AFTER!')
        return ret
    return inner

def myfunc():
    print('such wow!')
myfunc = shout(myfunc)
```

```
def shout(wrapped):
    def inner(*args, **kwargs):
        print(‘BEFORE!’)
        ret = wrapped(*args, **kwargs)
        print(‘AFTER!’)
        return ret
    return inner

def myfunc():
    print(‘such wow!’)
myfunc = shout(myfunc)
```

Good decorators are versatile

```
def shout(wrapped):
    def inner(*args, **kwargs):
        print('BEFORE!')
        ret = wrapped(*args, **kwargs)
        print('AFTER!')
        return ret
    return inner

def myfunc():
    print('such wow!')
myfunc = shout(myfunc)
```

```
def shout(wrapped):
    def inner(*args, **kwargs):
        print('BEFORE!')
        ret = wrapped(*args, **kwargs)
        print('AFTER!')
        return ret
    return inner

def myfunc():
    print('such wow!')
myfunc = shout(myfunc)
```

`*args` and `**kwargs` together

take any number of positional and/or keyword arguments `*args` is a list

`**kwargs` is a dictionary

```
def shout(wrapped):
    def inner(*args, **kwargs):
        print(‘BEFORE!’)
        ret = wrapped(*args, **kwargs)
        print(‘AFTER!’)
        return ret
    return inner

def myfunc():
    print(‘such wow!’)
myfunc = shout(myfunc)
```

```
def shout(wrapped):
    def inner(*args, **kwargs):
        print(‘BEFORE!’)
        ret = wrapped(*args, **kwargs)
        print(‘AFTER!’)
        return ret
    return inner

def myfunc():
    print(‘such wow!’)
myfunc = shout(myfunc)
```

`*args`

becomes

`args[0], args[1], args[2], …`

`**kwargs`

becomes

`key0=kwargs[key0], key1=kwargs[key1],` …

```
def shout(wrapped):
    def inner(*args, **kwargs):
        print('BEFORE!')
        ret = wrapped(*args, **kwargs)
        print('AFTER!')
        return ret
    return inner

def myfunc():
    print('such wow!')
myfunc = shout(myfunc)
```

This decorator is still wrong.

```
>>> myfunc.__name__
'inner'
```

```
def shout(wrapped):
    def inner(*args, **kwargs):
        print('BEFORE!')
        ret = wrapped(*args, **kwargs)
        print('AFTER!')
        return ret
    inner.__name__ = wrapped.__name__
    return inner

def myfunc():
    print('such wow!')
myfunc = shout(myfunc)
```

```
>>> import inspect
>>> inspect.getargspec(myfunc)
ArgSpec(args=[], varargs='args',
keywords='kwargs', defaults=None)
```

Graham Dumpleton created `wrapt`

http://bit.ly/decorators2014

```
import wrapt

@wrapt.decorator
def pass_through(wrapped, instance, args, kwargs):
    return wrapped(*args, **kwargs)

@pass_through
def function():
    pass
```

```python
import wrapt

@wrapt.decorator
def pass_through(wrapped, instance, args, kwargs):
    return wrapped(*args, **kwargs)

@pass_through
def function():
    pass
```

What about decorators with arguments?

`@unittest.skipIf()`

(simplified)

```
def skipIf(conditional, message):
  def dec(wrapped):
    def inner(*args, **kwargs):
      if not conditional:
        return wrapped(*args, **kwargs)
      else #skipping
        print(message)
    return inner
  return dec

@skipIf(True, ‘I hate doge’)
def myfunc():
    print(‘very print’)
```

```
def skipIf(conditional, message):
  def dec(wrapped):
    def inner(*args, **kwargs):
      if not conditional:
        return wrapped(*args, **kwargs)
      else #skipping
        print(message)
    return inner
  return dec

def myfunc():
    print(‘very print’)
myfunc = skipIf(True, ‘I hate doge’)(myfunc)
```

```
def skipIf(conditional, message):
  def dec(wrapped):
    def inner(*args, **kwargs):
      if not conditional:
        return wrapped(*args, **kwargs)
      else #skipping
        print(message)
    return inner
  return dec

def myfunc():
    print('very print')
myfunc = skipIf(True, 'I hate doge')(myfunc)
```

```
def skipIf(conditional, message):
  def dec(wrapped):
    def inner(*args, **kwargs):
      if not conditional:
        return wrapped(*args, **kwargs)
      else #skipping
        print(message)
    return inner
  return dec

def myfunc():
    print(‘very print’)
myfunc = skipIf(True, ‘I hate doge’)(myfunc)
```

```
def skipIf(conditional, message):
  def dec(wrapped):
    def inner(*args, **kwargs):
      if not conditional:
        return wrapped(*args, **kwargs)
      else #skipping
        print(message)
    return inner
  return dec

def myfunc():
    print('very print')
myfunc = skipIf(True, 'I hate doge')(myfunc)
```

```
def skipIf(conditional, message):
  def dec(wrapped):
    def inner(*args, **kwargs):
      if not conditional:
        return wrapped(*args, **kwargs)
      else #skipping
        print(message)
    return inner
  return dec

def myfunc():
    print(‘very print’)
myfunc = skipIf(True, ‘I hate doge’)(myfunc)
```

```
def skipIf(conditional, message):
  def dec(wrapped):
    def inner(*args, **kwargs):
      if not conditional:
        return wrapped(*args, **kwargs)
      else #skipping
        print(message)
    return inner
  return dec

def myfunc():
    print(‘very print’)
myfunc = skipIf(True, ‘I hate doge’)(myfunc)
```

```
def skipIf(conditional, message):
  def dec(wrapped):
    def inner(*args, **kwargs):
      if not conditional:
        return wrapped(*args, **kwargs)
      else #skipping
        print(message)
    return inner
  return dec

def myfunc():
    print(‘very print’)
myfunc = skipIf(True, ‘I hate doge’)(myfunc)
```

```
def skipIf(conditional, message):
  def dec(wrapped):
    def inner(*args, **kwargs):
      if not conditional:
        return wrapped(*args, **kwargs)
      else #skipping
        print(message)
    return inner
  return dec

def myfunc():
    print(‘very print’)
myfunc = skipIf(True, ‘I hate doge’)(myfunc)
```

```
def skipIf(conditional, message):
  def dec(wrapped):
    def inner(*args, **kwargs):
      if not conditional:
        return wrapped(*args, **kwargs)
      else #skipping
        print(message)
    return inner
  return dec

@skipIf(True, ‘I hate doge’)
def myfunc():
    print(‘very print’)
```

What about with `wrapt`?

```
import wrapt

def with_arguments(myarg1, myarg2):
    @wrapt.decorator
    def wrapper(wrapped, instance, args, kwargs):
        return wrapped(*args, **kwargs)
    return wrapper

@with_arguments(1, 2)
def function():
    pass
```

```
import wrapt

def with_arguments(myarg1, myarg2):
    @wrapt.decorator
    def wrapper(wrapped, instance, args, kwargs):
        return wrapped(*args, **kwargs)
    return wrapper

@with_arguments(1, 2)
def function():
    pass
```

```
import wrapt

def with_arguments(myarg1, myarg2):
    @wrapt.decorator
    def wrapper(wrapped, instance, args, kwargs):
        return wrapped(*args, **kwargs)
    return wrapper

@with_arguments(1, 2)
def function():
    pass
```

A few last examples

Counting function calls

```python
def count(wrapped):
    def inner(*args, **kwargs):
        inner.counter += 1
        return wrapped(*args, **kwargs)
    inner.counter = 0
    return inner


@count
def myfunc():
    pass
```

```
def count(wrapped):
    def inner(*args, **kwargs):
        inner.counter += 1
        return wrapped(*args, **kwargs)
    inner.counter = 0
    return inner


@count
def myfunc():
    pass
```

```
>>> myfunc()
>>> myfunc()
>>> myfunc()
>>> myfunc.counter
3
```

```
import time
def timer(wrapped):
    def inner(*args, **kwargs):
        t = time.time()
        ret = wrapped(*args, **kwargs)
        print(time.time()-t)
        return ret
    return inner

@timer
def myfunc():
    print(‘so example!’)
```

```
import time
def timer(wrapped):
    def inner(*args, **kwargs):
        t = time.time()
        ret = wrapped(*args, **kwargs)
        print(time.time()-t)
        return ret
    return inner

@timer
def myfunc():
    print('so example!')
```

```
>>> myfunc()
so example!
4.0531158447265625e-06
```

Thanks!

Colton Myers
@basepi
colton.myers@gmail.com

http://bit.ly/dec-pycon-2014